# Introduction to Programming
# Block Tutorial C/C++

Michael Bader
Lehrstuhl Informatik V
bader@in.tum.de

March, 4th – March, 8th, 2002

**Abstract**

This is where an abstract might go if you want one. There is usually not a lot of room for much here.

# C/C++ Tutorial – Overview

- From Maple to C
- Variables, Operators, Statements
- Functions: declaration, definition, parameters
- Arrays and Pointers
- From Pointers to Dynamical Data Structures
- Object Oriented Programming: From C to C++

# Wednesday Lesson – Overview

- defining Arrays
- Passing arrays as parameters
- How are arrays treated in C?
- Arrays of variable length: malloc, free, sizeof
- Pointers and arrays
- Arrays of characters: strings
- Multidimensional arrays

# Defining arrays in C

**Syntax:**

```
<type> <name>[<size>];
<type> <name>[<size1>][<size2>]...;

<type> <name>[] = <array_init>;
```

**Examples:**

```
char alphabet[26];
int primes[] = {2,3,5,7,11,13,17,19,23,29,31};
double matrix[10][10];
```

# Defining arrays in C

When defining an array, its size has to be specified:

- the size can be computed by the compiler if the array is initialized in the definition
- an explicit specification of the size defines an array of "empty" variables; the array elements are not initialized
- both, an explicit size and an initialising vector, may be given:

  ```
  int primes[11] = {2,3,5,7,11,13,17,19,23,29,31};
  ```

  The array may have more element than the initialization vector.
- the array size must not be given by a variable or expression

# How to access array elements

Array elements are accessed like in Maple, e.g. `alphabet[5]`, `primes[3]`

**Example Program:**

```
int main() {
   int i;
   int primes[] = {2,3,5,7,11,13,17,19,23,29,31};
   for(i=0;i<11;i++)
      printf("%d-th prime is %d\n", i, primes[i] );
}
```

**Remember:**

- **Array indices always start with 0**
- **Array size always states the number of elements**

# Arrays as parameters

**Example:**

```
int sum(int numbers[], int first, int last) {
   int i;
   int sum=0;
   for(i=first;i<=last;i++)
      sum += numbers[i];
   return sum;
}
int main(int argc, char** argv) {
   int primes[] = {2,3,5,7,11,13,17,19,23,29,31};
   printf("%d\n", sum(primes,0,10) );
}
```

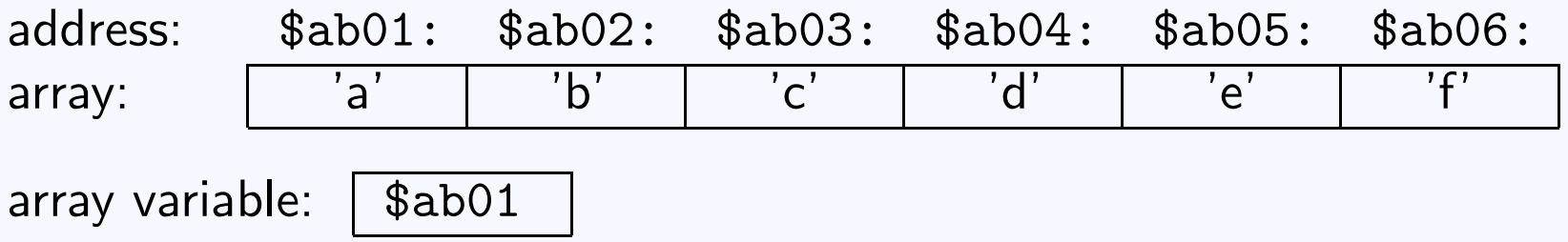
# Arrays as parameters (2)

**Another example:**

```
int strangesum(int numbers[], int first, int last) {
   int i, sum=0;
   for(i=first;i<=last;i++) {
      sum += numbers[i]; numbers[i]--;
   }
   return sum;
}
int main(int argc, char** argv) {
   int primes[] = {2,3,5,7,11,13,17,19,23,29,31};
   int s;
   do { s=strangesum(primes,0,10); printf("%d\n",s);
   } while (s>0) ;
}
```

# How are arrays treated in C?

- logically, an array is a sequence of variables
- physically, an array is a consecutive range of memory cells
- the variables are stored consecutively in the memory cells
- the array variable holds the beginning of the area of memory, i.e. the address of the first array element

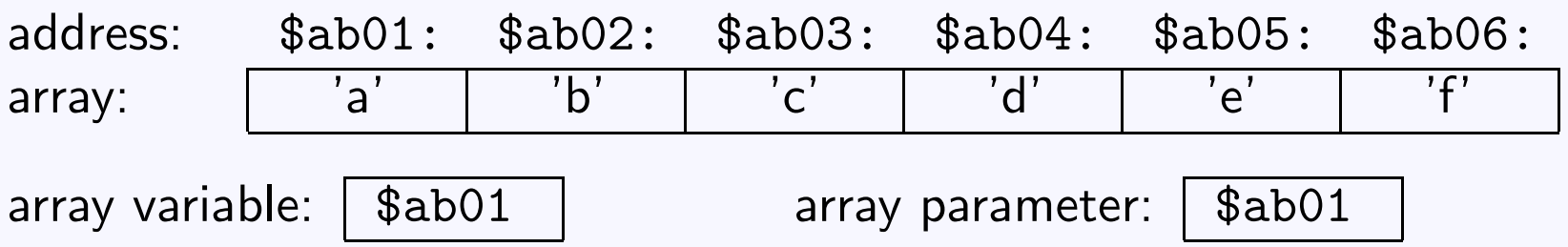



- no info on the size of an array is stored

$\Rightarrow$ store the size of each array **separately**

# Arrays and "call by value"

- when passing arrays as parameters, the value of the address of the first array element is passed to the parameter
- therefore, the parameter is (logically) a different array but has (physically) the same array elements




⇒ Changing elements of parameter array changes elements of original array!

⇒ **array elements can be changed permanently by a function!**

# A First Look at Pointers

- we've seen that, basically, an array is handled as an address of a certain variable
- C has a set of special types for such addresses: **pointers**

$\Rightarrow$ a **pointer** is an address in memory.

$\Rightarrow$ **pointers** have different types corresponding to the types of the variables that are stored at the respective address

**Examples:**

```
int* a;
double* f;
char* c;
```

# Pointers and Arrays

In C, pointers and arrays are close relatives.
For example, an array can be passed to a pointer parameter:

```
int sum(int* numbers, int first, int last) {
   int i, sum=0;
   for(i=first;i<=last;i++) sum += numbers[i];
   return sum;
}
int main() {
   int primes[] = {2,3,5,7,11,13,17,19,23,29,31};
   printf("%d\n", sum(primes,0,10) );
}
```

Obviously, the []-operator can also be applied to pointers!

# Arrays of Varying Size

When defining arrays, we always have to specify the size of it:

- Either by setting the size explicitly,
- or by initialising the array with the respective values

We can not define the size of an array using the value of a variable!

$\Rightarrow$ we could use pointers instead

**Problem:**

- an array variable holds an address of a suitable(!) area of memory
- how do we get such an area of memory for our pointer?

# Arrays of Varying Size – malloc()

In the library `stdlib.h`, there is a function to allocate memory: `malloc`

`malloc(n)` allocates an area of `n` bytes of memory and returns the address of the first memory cell.

**Example:**

```
main() {
    char i;
    char* alphabet;
    alphabet = malloc(26);
    for (i=0;i<26;i++) alphabet[i] = 'a'+i;
}
```

**Exercise:** Use an `int`-array in the example above!

# malloc() and integers

**Example:**

```
main() {
   int i=26;
   int* integers;
   integers = malloc(i);
   for (i=i-1;i>=0;i--) {
      integers[i] = i+1; printf("%d\n",integers[i]);
   }
}
```

This program seems to work, at first. However, if we replace 'i=26' by 'i=5000' (values depend on compiler and architecture) it fails:
The operating system will cast a memory access error.

# malloc() and sizeof()

**Problem:** different types require different amount of bytes in memory

**Solution:** C provides a special operator: `sizeof()`

**Example:**

```
main() {
   int i=26;
   int* integers;
   integers = malloc( i*sizeof(int) );
   for (i=i-1;i>=0;i--) {
      integers[i] = i+1; printf("%d\n",integers[i]);
   }
}
```

With most compilers, this program will work OK.

# `void` Pointers

- we have assigned `malloc`'s results to both `int*` and `char*` pointers
- So, what exactly is `malloc`'s return type?

**Answer:** `void* malloc( ...)`

`void` is a very special type. It may only be used

- in function definitions: `void name(void)`
- and for pointers: `void*`

A `void*` pointer is "just an address", there's no type associated with it.

$\Rightarrow$ We have to **cast** the pointer to the correct type

In the previous example, C did this implicitly!

# `void` Pointers and Explicit Casting

Normally, C compilers cast the `void*` pointers to the correct variable.

However, it is good style to do this cast explicitly:

```
integers = (int*) malloc( i*sizeof(int) );
```

Complete Program:

```
main() {
   int i=26;
   int* integers;
   integers = (int*) malloc( i*sizeof(int) );
   for (i=i-1;i>=0;i--) {
      integers[i] = i+1; printf("%d\n",integers[i]);
   }
}
```

# Arrays of Characters – Strings

In C, a string is an array of characters, terminated by the 0-character:

| string: | 'a' | 'b' | 'c' | 'd' | 'e' | '\0' |
|---|---|---|---|---|---|---|

**Example:**

```
main() {
    int i=0;
    char* text = "The Quick Brown Fox Jumps Over the Lazy Dog.";
    printf("%s\n",text);

    while (text[i]!='\0') printf("%c",text[i++]);
    printf("\n");
}
```

# String Operations

The library <string.h> provides a set of functions to manipulate strings:

```
char* strcat(char*, const char*);
char* strcpy(char*, const char*);
char* strlen(const char*);
char* strcmp(const char*, const char*);
```

**Beware:** Make sure that modified strings have ample amount of memory!

**Example:**

```
char hello[100] = "Hello ";
const char world[] = "World!";
strcat(hello,world);
printf("%s\n",hello);
```

# A Few Remarks on `const` Types

**Syntax:** `const <type>`

`const` can be put ahead of any type specifier:
Variables declared using a `const` type may not be modified!

**Examples:**

```
const double pi = 3.14159265;
const int answer = 42;
const char space = ' ';
const char login[] = "bader";
```

Most Compilers only issue a warning, though.

# Multidimensional Arrays of Fixed Size

We already introduced how to define multidimensional arrays of fixed size:

```
main() {
   int i,j;
   char tictactoe[3][3] = {{'x','x','o'},
                           {'o','x','o'},
                           {'o','x','x'}};
   for(i=0;i<3;i++) {
      for(j=0;j<3;j++) printf("%c",tictactoe[i][j]);
      printf("\n");
   }
}
```

# Multidimensional Arrays of Dynamic Size

Again, to define arrays of dynamic size, we have to use pointers and `malloc`:

Idea: define an array of arrays (of arrays of arrays . . . )

**Example:**

```
main() {
   int i,size=3;
   char** tictactoe;
   tictactoe = (char**) malloc(size*sizeof(char*)) ;
   for(i=0;i<size;i++)
      tictactoe[i] = (char*) malloc(size*sizeof(char)) ;
}
```

# Multidimensional Arrays – Memory Layout

**Example:** Tic Tac Toe

| address: | $ab01 | $ab02 | $ab03 |
|---|---|---|---|
| t: | t[0]=$ab10 | t[1]=$ab35 | t[2]=$ab47 |

| address: | $ab10 | $ab11 | $ab12 | $ab13 |
|---|---|---|---|---|
| t[0]: | 'x' | 'x' | 'o' | '\0' |

| address: | $ab35 | $ab36 | $ab37 | $ab38 |
|---|---|---|---|---|
| t[1]: | 'o' | 'x' | 'o' | '\0' |

| address: | $ab47 | $ab48 | $ab49 | $ab4a |
|---|---|---|---|---|
| t[2]: | 'o' | 'x' | 'x' | '\0' |

# `main()` revisited

"Correct" way to define main: `int main(int argc, char** argv)`

- `argv` is an array of strings (containing the command line parameters);
- `argc` is the size of the array;
- `argv[0]` is the name of the program.

**Example:** print all command-line parameters

```
int main(int argc, char** argv) {
   int i;
   for(i=0;i<argc;i++)
      printf("%s\n",argv[i]) ;
}
```